A Performance Analysis of Java and C

Ambika PAJJURI and Haseeb AHMED

***Abstract* – Java and C are two popular specification languages used to define systems of all sizes and forms. In this paper, we present a performance comparison of various algorithms written in C and Java on Windows and Linux environments. The metrics considered in the analysis include speed of execution, memory usage, Java vs. C overheads and other special features that characterize the two languages. The paper presents a survey of work in this area and a discussion their results. In our proposed project plan, we intend to investigate both languages based on how their design choices influence their performance rather than by semantics and programming paradigms. The algorithms for the analysis will be chosen to represent those commonly used in embedded systems (such as FIRs) as well as more exotic ones like the MD5 cipher.**

***Index Terms*-Java, C, Performance comparison**

I. Introduction

THE design of a computer language often results from a desire to solve a set of problems in a given domain. Most modern languages strive to be the 'one size fits all' type of solution implying a broad set of goals. These often-divergent goals often lead to a 'specialization' effect wherein certain features are readily adopted into the mainstream and others fade away. We compare two such languages - Java and C.

Conventional wisdom suggests that Java and C make an odd pair to investigate. They do not share a common programming paradigm (object oriented vs. procedural). Moreover, Java tries to insulate users from the underlying architecture, while C is very accommodating to low-level access. It is perhaps for this reason that much of the published research work has focused on more natural comparisons such as Java and C++.

Java and C are both specification languages. C was conceived as a 'high level assembly' language whereas Java had its roots as an embedded/portable language for set-top boxes. The C language derived much of its semantics from its ancestor B, and so a simple procedural pass-by-value methodology was adopted. Java, due to its (very lucrative) requirement for portability and ease of use, chose to go with an object-oriented model. So while Java's internals grew to be more complex, the programmer was largely insulated from all the details.

Both Java and C have design choices that were intended to aid the programmer and (or) the compiler. Many of these features remain unused or unimplemented despite underlying hardware support. For example, hardware often has support for execution of MAC type instructions but there is no direct syntax for doing so in C or Java.

C allows a lot of flexibility to the programmer, but it is left largely to programmers and compilers to exploit these features. In the case of Java, the Java Virtual Machine (JVM), on which all Java programs run, hides many of the optimizations. Java, in its current form, is not very suitable for use in embedded systems. This is because does not support operations like direct memory access, interrupt handling and scheduling to meet hard deadlines.

The rest of this paper is organized as follows – In Section II, we present a summary of related work in this field. In Section IV, we discuss our project plan.

II. Related Work

A. The Java Performance Report – Osvaldo Pinali Doederlein

The Java Performance report [1] compares the performance of C and Java algorithms on Win32 platforms. The tests used a suite of algorithms written in C (BYTEmark) and their direct port to Java (JBYTEmark). The results presented in the paper indicate that, in general, C outperformed Java, as one would expect. However, the performance of Java depended on the underlying JVM, and also the specific algorithm under test. In fact, in some algorithms, specific Java implementations (especially IBM's JDK 1.3) outperformed C.

B. Binaries Vs. Bytecodes - Chris Rijk.

The results from [1] were further strengthened by [2] where IBM's JDK v1.3.0 was seen to outperform even Microsoft's Visual C compiler in many of the benchmarks, as shown in Fig.1. This challenges the notion that the JVM is always an extra piece of luggage.

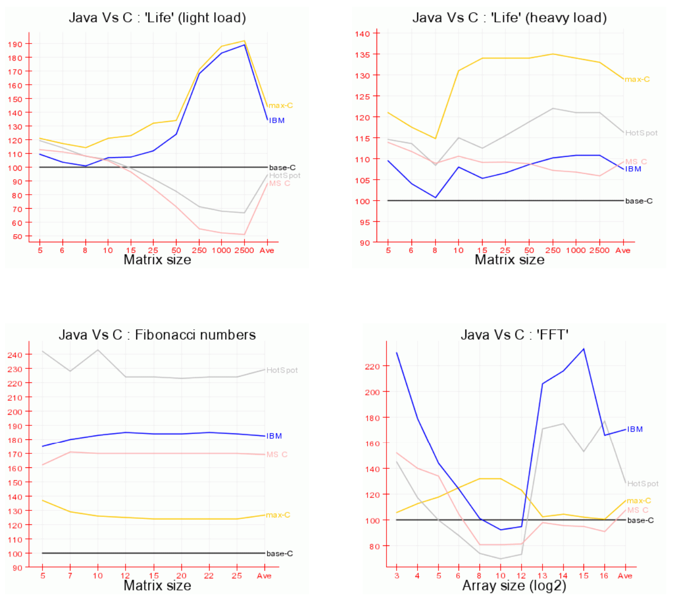

Figure 1: Comparison results between IBM's JVM and C. From [2] Chris Rijk, Binaries Vs Byte-Codes

C. The Java Performance Analysis for Scientific Computing – Roldan Pozo

In contrast to [1], Pozo considered a more diversified array of algorithms commonly used in scientific computing. His approach was unique in that he worked with operations that were both CPU and memory intensive (*e.g.,* large matrix (1000x1000) multiplication operations). His observations were as follows:

C's strengths:

- Allows for direct mapping to hardware
- Provides more opportunities for optimizations
- No penalty for garbage collection

Java's strengths:

- Performance varies widely by the choice of a JVM – the best results were from IBM and Sun.
- Performance closely linked to underlying hardware (*i.e.* faster CPU does make an impact)

Pozo's experiments also showed that unlike the performance of C/C++ compilers, there is a lot of variation in the performance of the different JVMs. The application of some small non-standard optimization also produced significant benefits (as shown in Fig. 2). Considering the benefits incurred, such optimization should probably become the norm.

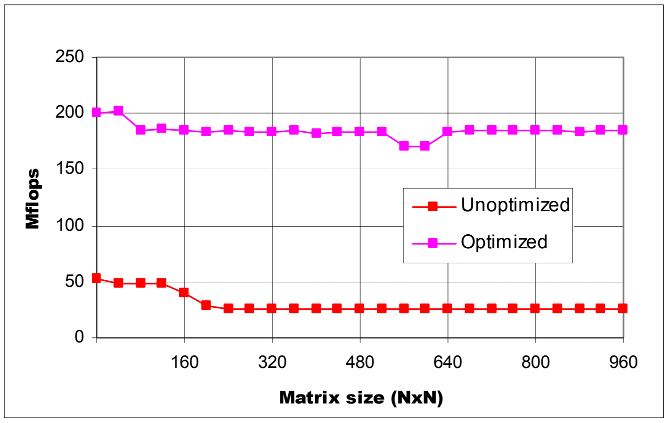

Figure 2 Results of matrix multiplication from R. Pozo [3] showing that select matrix optimizations can yield significant improvements

R. Pozo concluded with two important comments:

i. Java requires more aggressive memory mechanisms to compensate for the gawkiness of automatic garbage collection. (This point is reinforced by [3]).
ii. JVMs are increasingly important in byte-code manipulation. (Also see [6], [7] for more recent research).

Other more subtle issues alluded to why Java was less favorable than C for use in large scientific and engineering applications. These include the lack of efficient multidimensional arrays, the inability to take advantage of fused multiply-add and associativity operations in compiler optimizations (also confirmed by [8]).

D. The Java Real-time Extension Specification

Another emerging area for study is the Java 'Real-time Extension Specification' [5]. It is expected to bring long desired advantages of the Java Platform, like binary portability, dynamic code loading, tool support, safety, security, and simplicity, to an important industry segment: real-time systems. This extension targets both "hard real-time" and "soft real-time" systems. The specification addresses many issues, including garbage collection semantics, synchronization, thread scheduling, JVM-RTOS interface, and high-resolution time management.

III. Project Plan

Our project plan is to expand on the existing work with the following strategy:

A. Run a gamut of algorithms.

a. The suite of algorithms we intend to use will be both CPU and memory intensive. For example, we would consider FIR variants that are closer to traditional embedded operations. If time permits, we plan to analyze exotic algorithms such as the MD5 cipher.
b. Some of the other algorithms under consideration are –
   i. Simple Fast Fourier Transforms.
   ii. BYTEmark & JBYTEmark.
   iii. Matrix addition, multiplication & dot products (over varied sizes).
   iv. Miscellaneous: Adler32, MD5.

(The eventual list may vary depending on implementation constraints)

B. Evaluate the results over Linux and Microsoft Windows operating systems.

a. Linux and Windows have distinct architectures. This extenuates C and Java's design where C likes to be close to the native OS while Java relies on its JVM.

   i. Memory management is one of the key differences between Java and C. We want to expand on the work from [4] and to identify other such opportunities for enhancements to both Java and C.
   ii. Another area that has not been well investigated is the primitive data type selection in Java. Strings in particular pose a challenge because they consist of 16-bit Unicode. It remains to be seen how this choice affects garbage collection.

b. Compiler Optimizations offer another interesting area that affects performance.

   i. Java is unique in its run-time optimizations. This methodology however is unproven – especially in embedded systems. C on the other hand provides constructs like pointers that allow very close interaction with the underlying architecture.
   ii. Java also adds features such as automatic bounds checking. This feature is entirely missing in C.

This 'feature' is shown to be detrimental with no way to turn it off.

C. *Winner takes all?*

Just as any one language cannot lay claim to solving all problem domains, our performance analysis will rate Java and C on different metrics. We hope this will aid in the selection of the right language for the right task and provide future opportunities for exploration.

a. Execution Speed – This factor is readily visible to programmers (and end users). Execution speed is often seen as the most important attribute of a language's performance.
b. Memory Usage – Is this a moot point in these times of cheap memory? We do not think so – especially since embedded systems have far more stringent memory requirements. Minimizing memory usage is becoming increasingly important, as expensive (and slow) I/O is still the bottleneck, even with faster CPUs.
c. Language Features – Java with its runtime optimizations, garbage collection and freebies like bounds checking seems very impressive. C on the other hand places the entire burden on the programmer. One of the unanswered questions is on the cost(s) of such extremes.

References

[1] Osvaldo Pinali Doederlein, *The Java Performance Report - Part III,* http://www.javalobby.org/fr/html/frm/javalobby/features/jpr/part3.html
[2] Chris Rijk, *Binaries Vs Byte-Codes, Ace's Hardware*, June 27, 2000.
[3] Roldan Pozo, *"Java Performance Analysis for Scientific Computing"* - National Institute of Standards and Technology, USA. This report was presented at the UKHEC: Java for High-end Computing in Nov 2000.
[4] Milo Martin, Manoj Plakal and Venkatesh Iyengar, *"Top-Level Data-Memory Hierarchy Performance: Java vs. C/C++"* - University of Wisconsin – Madison, Dec 1996.
[5] RTSJ - *Real-time Specification for Java*, http://www.javaseries.com/rtj.pdf
[6] Kyle R. Bowers, David Kaeli, *"Characterizing the SPEC JVM98 Benchmarks on the Java Virtual Machine"* – Northeastern University Computer Architecture Research Group.
[7] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio and J. Sabarinathan, *"Java Runtime Systems: Characterization and Architectural Implications"*. A preliminary version of this paper appeared in the International Conference on High Performance Computers and Architectures (HPCA-6), Feb 2001.
[8] S.P. Midkiff, J.E. Moreira and M. Snir, *"Optimizing Array Reference Checking in JAVA programs"*, IBM Systems Journal, 37 (409-453) 1998.